Apprentissage de proprietes semi-globales sur un systeme distribue

Ivan Lavallee

INRIA

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 819

APPRENTISSAGE DE PROPRIETES SEMI-GLOBALES SUR UN SYSTEME DISTRIBUE

Ivan LAVALLEE

AVRIL 1988

APPRENTISSAGE DISTRIBUE DE LA TERMINAISON ET D'AUTRES PROPRIETES STABLES D'UN PROCESSUS

APPRENTISSAGE DE PROPRIETES SEMI-GLOBALES SUR UN SYSTEME DISTRIBUE

DISTRIBUTED LEARNING OF TERMINATION AND OTHER STABLE PROPERTIES

Ivan LAVALLEE

Résumé :

Ce rapport étudie le problème de l'apprentissage de sa propre terminaison par un processus d'un système opératoire distribué, par rapport à l'algorithme exécuté. On montre comment ce problème se pose en termes de point fixe, et comment on en déduit un algorithme général qui permet, entre autres choses d'acquérir des informations semi-globales sur la topologie du graphe (virtuel) des communications inter-processus lors de l'exécution d'un algorithme, ce graphe pouvant être quelconque (mais connexe). On retrouve des algorithmes bien connus lorsqu'on spécialise notre algorithme aux cas classiquement étudiés (terminaison sur un anneau, un arbre, etc...).

Mots clés : *Apprentissage, terminaison, point fixe, composantes connexes.*

Abstract :

In the following we show how a process in a distributed operating system can learn its own termination. We show that there is a general formulation in terms of fixed point equation. And we give a general algorithm which permit to learn semi-global topological properties on the distributed network.

Key words : *Learning, termination, fixed point, connected components.*

I - INTRODUCTION

Dans un algorithme distribué, l'arrêt d'un processus n'implique pas nécessairement que l'algorithme général s'arrête. Le problème de la terminaison distribuée est un problème majeur en algorithmique distribuée. En fait il recouvre deux problèmes :

a - Détecter si tous les processus sont à l'arrêt

b - S'assurer qu'on a bien accompli le calcul voulu.

On remarquera que le fait que tous les processus soient arrêtés entraîne que l'algorithme distribué est terminé, mais n'assure absolument pas que l'arrêt de chaque processus se soit opéré à bon escient, et donc n'assure pas qu'on ait bien accompli le calcul voulu.

II - LE PROBLEME DE LA DETECTION DE LA TERMINAISON DISTRIBUEE

Le problème posé est de détecter quand un calcul distribué se termine. On peut décrire ce problème comme suit [TOP], [RAN], [CH,MI], [FRA].

Supposons donnés N processus P_i , $0 \leq i \leq N$, chacun d'entre eux étant localisé sur le sommet d'un graphe.

Considérons les processus qui sont en cours d'exécution d'un algorithme distribué, c'est à dire que chacun d'entre eux exécute un algorithme séquentiel et échange des messages avec les autres processus par l'intermédiaire d'un réseau de communication dont les canaux sont les arêtes (éventuellement les arcs) du graphe, Chaque processus peut être soit "actif" (il est en train d'exécuter une instruction de son algorithme propre) soit "inactif" (on choisit ce terme assez vague pour ne pas distinguer, du moins dans un premier temps, l'état où le processus a définitivement terminé, du cas où il est en attente).

Les règles de fonctionnement sont alors les suivantes :

1) Seuls les processus "actifs" peuvent envoyer des messages

2) Un processus ne peut passer de l'état "inactif" à l'état "actif" qu'à la suite de la réception d'un message. (au bout d'un temps plus ou moins long après la dite réception).

3) Un processus peut passer de l'état "actif" à l'état "inactif" à tout instant. (à condition qu'il ait reçu au moins un message).

Si tous les processus sont à l'arrêt, l'algorithme distribué ne peut plus continuer plus avant, et alors,

i) Soit il est terminé, il a effectué le calcul pour lequel il a été conçu
ii) Soit il est bloqué

On dit aussi parfois qu'il se termine suivant deux états ;

i) il est "bien" terminé, c'est à dire que le résultat pour l'obtention duquel l'algorithme a été conçu a *effectivement* été calculé ;

ii) il est "mal" terminé.

Résoudre le problème de la terminaison distribuée c'est permettre à tout processus de détecter qu'il a (bien) terminé, et que tous les processus atteignent cet état.

Plusieurs solutions ont été proposées pour résoudre ce problème. Toutes les solutions proposées commencent par structurer l'ensemble des processus de façon à pouvoir visiter chacun des processus du réseau pour vérifier la propriété de terminaison.

Les structures de base utilisées pour effectuer le parcours du graphe de communications sont l'anneau unidirectionnel ou circuit hamiltonien [DI,FE,GA], [RAN], ou encore un circuit eulérien [MIS], et également un arbre couvrant sur le graphe associé au réseau [TOP], [FR,RO]*), [MAT].

Les différentes techniques ci-dessus évoquées nécessitent de plus une stratégie de "visite" des différents processus (et éventuellement un algorithme de construction de la topologie de l'anneau ou du circuit lorsqu'ils sont virtuels).

En fonction de la topologie de contrôle construite sur le graphe, il est nécessaire de déterminer un mode de parcours.

Ainsi, si la structure de contrôle construite est un arbre couvrant, le mode de parcours utilisé est celui du calcul diffusant dont le principe a été introduit par Dijkstra et Scholten [DI,SC]. Lorsque la structure de contrôle est un anneau (ou un circuit) de processus, on utilise plutôt un jeton circulant [DI,FE,GA]. Dans l'algorithme de Topor [TOP], la topologie de contrôle sur le graphe est une arborescence, et on utilise aussi un système de jetons. Dans cet algorithme, le processus associé à la racine joue un rôle

---

*) On trouvera dans [LA2] et [LA,RO] un algorithme pour construire dynamiquement un tel arbre.

particulier, il lance la détection en émettant vers ses successeurs (ses fils), des jetons, chaque processus recevant un jeton réemettant lui même des jetons vers ses fils, et ainsi de suite... jusqu'aux feuilles. Dans [DI,FE,GA], on trouve une hypothèse supplémentaire qui est celle de l'existence de connections spécialisées uniquement dans le contrôle et l'envoi de signaux.*)

Les progrès récents en matière de supraconductivité mettent à l'ordre du jour des machines multiprocesseurs comportant plusieurs centaines de milliers, voir des millions de processeurs. Sur de tels systèmes, le système opératoire doit être complètement distribué, ne serait-ce que pour des raisons de résistance aux pannes.

La réallocation des ressources système (processeurs pour ce qui nous concerne) ne peut être le fait d'un processus maître, allant détecter l'état des autres processus en cours. Pour le problème de la terminaison, aller tester qu'un million de processus ont terminé serait prohibitif en termes de temps et de charge du réseau de communication (nombre de messages).

Par conséquent, nous allons nous intéresser au problème de l'apprentissage de sa propre terminaison par un processus.

L'étude s'articule sur deux parties. Dans la première partie, nous montrerons les conditions nécessaires de terminaison d'un processus, nous dirons également comment et quand elles sont suffisantes.

Nous allons donner des règles qui, sous certaines conditions permettent à tout processus de s'arrêter en sachant qu'il n'aura plus à intervenir dans l'algorithme distribué considéré. De plus, nous voulons que ces règles assurent bien que *tous* les processus s'arrêtent en un temps fini. Nous sommes donc concernés ici par des algorithmes distribués possédant la propriété dite de *fatalité.*

Dans la deuxième partie, nous donnerons un algorithme d'apprentissage de la termaison pour un processus d'un système distribué. Cet algorithme très général détecte les composantes connexes ou fortement connexes du graphe éventuellement associé aux communications de l'algorithme sur lequel on veut tester la terminaison.

En fait, il s'agit d'un algorithme général de détection d'un état stable pour un processus, et plus particulièrement de résolution de problèmes ressortissant à une formulation sous forme d'une équation de point fixe.

---

*) Cette hypothèse, non réaliste dans le cas de réseaux d'ordinateurs géographiquement éloignés les uns des autres, peut devenir réaliste dans le cas d'une machine distribuée.

lère PARTIE

I - CONCEPTS DE BASE ET NOTATIONS

I.1 - Le concept de processus

Le mot "processus" recouvre des acceptions différentes bien que voisines dans la littérature informatique. Du point de vue du système la notion de processus est, la plupart du temps, identifiée à celle de tâche, qu'on peut alors définir comme une unité indépendante d'exécution dans un système.

L'utilisation du terme processus permet de s'abstraire de la gestion des tâches proprement dite, permettant par là même de s'abstraire de particularités contingentes de tel système d'exploitation ou tel autre. Le terme de processus a été introduit par Dijkstra en 1968 [DIJ] pour, justement, étudier comment modéliser les rapports devant exister entre diverses unités d'exécution indépendantes devant se partager des ressources communes (tant matérielles que logicielles), et ce, de manière théorique, indépendamment des particularités des machines utilisées.

Pour ce qui nous concerne, nous allons utiliser le terme de processus avec une optique très proche de celle qui précède.

Nous allons, d'une manière informelle, considérer un processus comme une unité d'exécution élémentaire d'un algorithme distribué ou parallèle, plusieurs telles unités ou processus pouvant opérer simultanément, chacun étant indivisible.

C'est à dire que nous considérerons un processus comme étant intrinsèquement séquentiel du point de vue de son exécution. En conséquence, nous adopterons une vision statique du concept de processus, contrairement à d'autres auteurs. Ainsi, on peut considérer de façon intuitive que le concept de processus est une abstraction de la réalité physique qu'est un processeur.

D'une façon plus formelle, on peut dire qu'un processus est un algorithme séquentiel, déterministe ou non, (Machine de Turing par exemple) muni de primitives de communication lui permettant d'émettre et de recevoir des messages, c'est-à-dire de communiquer avec d'autres processus.

L'interaction entre deux ou plusieurs processus se fait alors uniquement par le moyen d'échanges de messages. Les primitives d'échange de messages sont fournies par un système sous-jacent - que nous nommerons le système Postal - dont les propriétés et caractéristiques varient d'un réseau distribué ou d'une machine parallèle à l'autre.

Un algorithme distribué est vu comme une collection de processus, indépendamment du langage de programmation utilisé. Chaque processus peut alors être défini par un ensemble d'états et un ensemble de transitions de la forme suivante (voir [CAR]) :

(ancien état, message reçu) → (nouvel état, message émis)

Intuitivement, on peut considérer la mémoire partageable d'une machine parallèle comme un processus particulier ne faisant que recevoir et émettre des messages sans leur faire subir aucun traitement.

I.2 - Le système postal

Nous appellerons système postal le système logique associé au médium de communication. Nous considérons qu'il s'agit d'un niveau du logiciel de communication - une couche - et nous supposerons que cette couche assure au système postal les propriétés suivantes :

1/ Tout message envoyé d'un processus P_i vers un processus P_j arrive en un temps fini mais non prévisible (propriété d'asynchronicité).

2/ L'intégrité de tout message est conservée par la transmission. Cela signifie qu'aucun message n'est ni perdu ni altéré par la transmission.

3/ Lorsque deux messages a et b sont envoyés dans un ordre déterminé, soit ab par un même processus P_i vers un même processus P_j, ces deux messages sont reçus dans l'ordre ab par le processus P_j.

On considère en particulier que le réseau n'a pas de panne, ou plutôt que si panne il y a, la couche de logiciel du système postal est capable de la prendre en compte de façon totalement transparente pour le niveau auquel nous nous intéressons.

I.3 - La spécification des algorithmes

Sans vouloir à proprement parler définir un langage pour algorithmes parallèles ou distribués ; nous avons voulu formaliser suffisamment la spécification de nos algorithmes pour lui conférer une forme concise et sans ambigüité

Les primitives que nous utiliserons pour spécifier nos algorithmes sont toutes, plus ou moins empruntées à CSP [HOA]. De même, qu'en séquentiel on spécifie des algorithmes en pseudo-Pascal ou autre, nous spécifierons ici en ce qu'on peut considérer comme quasi-CSP.

La différence essentielle avec C.S.P est que nous considérons des communications *non*-bloquantes. (On peut, en C.S.P classique, décrire la gestion de tampons de communication suivant les communications non bloquantes).

I.3.1 - Les primitives de communication

Nous noterons

P_x ?? < message >

la lecture d'un message de la file d'attente dans l'ordre d'arrivée (PAPS, ou FIFO en anglais), la présence du "x" en indice permettant de savoir quel est le processus qui a, le dernier, émis ce message.

On pourrait considérer une variante supprimant x, et on considérerait alors que le message est muni d'une en-tête contenant cette information.

L'émission d'un message quant à elle sera notée *)

P_x !! < message >

La valeur prise par x précisant à quel processus est destiné le message (en première instance).

Cela présuppose bien entendu que dans le graphe des communications, il existe un arc (ou une arête) entre le processus i courant et le processus x auquel on envoie le message.

Cette émission, au contraire de ce qui se passe en CSP est ici non bloquante, tout message envoyé à x étant reçu par celui-ci, au bout d'un temps fini dans un tampon.

---

*) Attention, il n'y a pas symétrie au niveau de la sémantique de ces spécifications.

I.3.2. Abréviations et facilités d'écriture

Notre propos étant essentiellement de spécifier des algorithmes, de la manière la plus concise possible, nous utiliserons un certain nombre d'abréviations lorsque celles-ci n'entrainent aucune ambiguité.

L'écriture suivante :

$$\forall\ i \in \text{tableau},\ P_i\ !!\ <\text{message}>$$

se lit : envoyer à tous les P_i tels que i soit dans l'ensemble "tableau" le message.

Une autre écriture équivalente serait :

$$\bigcup_x :\ x \in \text{liste}\ ,\ P_x\ !!\ <\text{message}>$$

On utilisera aussi des opérations ensemblistes du type :

$$TAB := TAB \cup VEC - \{x,y\}$$

qui signifie :

> Affecter à la variable TAB (qui désigne ici un tableau), l'ensemble TAB ∪ VEC dont on a retiré les éléments x et y.

De même, on utilisera des notations synthétiques du type :

$$\bigwedge_{\ell \neq j} \text{compl}(\ell) = 1 \rightarrow \ldots$$

ce qui pour la commande gardée considérée signifie (le tableau compl étant booléen ou logique) que si tous les éléments de compl, sauf le j-ième sont à 1, la garde est "vraie" (le booléen associé est 1).

II - EQUATION DE POINT FIXE ASSOCIEE A LA TERMINAISON

Lorsque tous les processus ont terminé, si le réseau est muni d'un arbre couvrant de contrôle, en considérant l'arborescence induite par une relation père-fils entre les sommets de l'arbre, la détection de la terminaison générale par la racine ainsi induite se fera simplement.

Pour un sommet quelconque x_i de l'arborescence, on a :

état(x_i) $\in \{0,1\}$ avec

$$\text{état}(x_i) = \begin{cases} 1, & x_i \text{ a terminé} \\ 0 & \text{sinon} \end{cases}$$

En notant T_{x_i} l'ensemble des fils de x_i, on a la relation

$$\text{état}(x_i) = \text{état}(x_i) \cdot \prod_{x_j \in T_{x_i}} \text{état}(x_j)$$

qui est une équation de point fixe représentative de l'état terminal ou non.

Il suffit alors que :

- Le processus P_i associé à x_i envoie son message de détection de terminaison à son père P_j associé à x_j dès lors que état $(x_i) = 1$
- Tout processus P_i associé à une feuille de l'arborescence détecte sa propre terminaison en un temps fini. En effet, lorsque tous les processus associés aux feuilles de l'arborescence ont détecté leur propre terminaison, il suffit de considérer le nouvel arbre déduit du précédent par suppression de ces feuilles. Ainsi, si chacun des processus associés aux feuilles termine en un temps fini, au bout d'un temps fini, l'arborescence va se trouver réduite à une feuille (la racine de l'arborescence). Le processus associé à cette feuille détectant sa propre terminaison en un temps fini, tout l'algorithme distribué sera alors terminé, et ce en un temps fini.

Il nous faut donc mettre en évidence les conditions nécessaires de terminaison de chaque processus pris individuellement, la terminaison globale de l'algorithme distribué étant conditionnée par l'arrêt des différents processus pris individuellement qu'il met en oeuvre, c'est-à-dire les conditions qui font que

$$\text{état}(x_i) \cdot \prod_{x_j \in T_{x_i}} \text{état}(x_j) = 1 \text{ lorsque } \prod_{x_j \in T_{x_i}} \text{état}(x_j) = 1.$$

<u>Remarque :</u>

Cette relation de point fixe est vraie pour <u>tout</u> processus x_i, T_{x_i} (voir [LA1]) représente alors la sous arborescence de racine x_i issue de l'arborescence couvrante de contrôle.

III - CONDITIONS NECESSAIRES POUR LA TERMINAISON DISTRIBUEE.

Dans un réseau distribué, dans la mesure où chaque processus local concourt à la résolution d'un problème global, l'algorithme distribué ne saurait, en général, se réduire à une simple juxtaposition d'algorithmes séquentiels locaux. L'exécution d'un algorithme séquentiel localisé en un noeud est donc en général, conditionné par les messages échangés. Lorsqu'un noeud n'a plus ni à émettre ni à recevoir de messages avec d'autres noeuds du réseau, le processus afférant peut être considéré comme hors du réseau distribué.

La terminaison pour un tel processus sera assurée par son propre algorithme, indépendamment des valeurs prises par les variables locales des autres processus. Ceci nous permettra de dire que le processus va se terminer (sous l'hypothèse implicite que son algorithme local lui même se termine)*).

En plus des propriétés décrites au Chapitre 0 pour le réseau de communication, nous supposerons ici que le graphe considéré est orienté, c'est à dire que les lignes de communication sont unidirectionnelles.**)

Sur un arc (x,y), on peut envoyer un message de x vers y, mais pas de y vers x.

De plus on supposera pour chaque processus associé à un noeud,

i) qu'il connait l'identité de chacun des noeuds auxquels il est connecté par un arc. S'il ne la connaît pas au début de l'algorithme, on peut considérer qu'il connaît l'existence des noeuds, et qu'il en connaîtra l'identité lorsqu'il en recevra un message.

ii) Pour chaque noeud auquel il est connecté par un arc, il sait s'il s'agit d'un "père" ou d'un "fils".

iii) Tous les noeuds ont un identificateur unique. Si I(i) et I(j) sont les identificateurs des processus i et j, alors on a :

$$i \neq j \Leftrightarrow I(i) \neq I(j)$$

Cette dernière condition est assez forte, elle peut donner lieu à un prétraitement pour qu'il en soit ainsi. Il existe des algorithmes pour ce faire (voir [BOU]), la description d'un tel algorithme est hors de notre propos.

---

*) Cette hypothèse convient pour le présent travail, mais elle admet en général une exception de taille, celle des processus des centraux téléphoniques qui, par définition ne s'arrêtent jamais.

**) Nous prenons ce modèle car il contient le modèle non orienté dans lequel on peut toujours remplacer les arêtes, chacune par une paire d'arcs orientés de sens contraire.

III.1 - Processus fermé, complet, terminé

Definition 1 :

On dira qu'un processus est *complet* par rapport à un algorithme distribué, lorsque les valeurs prises par ses variables,*) ne sont plus susceptibles de modification, suite à la réception d'un message par le processus.

C'est-à-dire quand un tel processus ne cherchera plus à lire de message (i.e. il n'exécute plus d'ordre de lecture).

Définition 2 :

Un processus complet par rapport à un algorithme distribué, ou par rapport à une phase de celui-ci, sera dit *fermé* lorsqu'il n'aura plus à exécuter dans son algorithme séquentiel d'instruction d'envoi de message vers un autre processus. (i.e. lorsqu'il n'aura plus pour des raisons logiques, à envoyer de message à l'un de ses voisins).

Remarque :

On peut considérer un processus fermé comme étant réduit à une partie de son algorithme séquentiel, la partie ne contenant pas de traitement de message, l'émission de message étant elle même considérée comme une action séquentielle comme une autre. Il y a alors identification de la partie de l'algorithme proprement dit, et du processus.

Définition 3 :

Un processus fermé sera dit *terminé* lorsque la partie de l'algorithme séquentiel à laquelle il est réduit sera terminée.

Un processus qui n'est pas complet sera dit "en cours", un processus qui n'est pas fermé sera dit "ouvert", et un processus qui n'est pas terminé sera dit "vivant".
Le graphe de transition des états possibles d'un processus est alors :

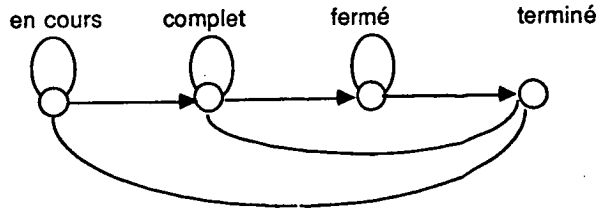


Figure 1

*) Locales, par définition.

III.2 - Conséquences des définitions :

Dans un réseau distribué représenté par un graphe, un processus associé à un sommet x du graphe, tel que

$\Gamma^-(x) = \phi$ (rappel : dans un graphe orienté G=(X,U), on note $\Gamma^+(x)=\{y|(x,y):y\in X \text{ et } (x,y)\in U\}$, et $\Gamma^-(x)=\{y|(y,x):y\in X \text{ et } (y,x)\in U\}$

est complet (évident).

Un processus dont tous les prédécesseurs (les pères) sont fermés est complet.

En effet si tous les prédécesseurs d'un processus X sont fermés, ils ne peuvent plus envoyer de message à X, par conséquent X ne peut plus recevoir de message, donc les valeurs prises par ses variables ne sont plus influencées par les messages reçus, le processus est complet*).

Tous les algorithmes séquentiels dont il est mention ici se terminent en un temps fini.

Un processus X associé à un sommet x du graphe tel que :

$\Gamma^+(x) = \phi$

est fermé dès lors qu'il est complet.

Remarque :

Un processus terminé est à la fois fermé et complet et un processus fermé est complet, mais un processus peut passer directement de l'état en cours, ainsi que de l'état complet à l'état terminé (voir Fig. 1), ceci en fonction de la logique de son programme propre ou des gardes (voir chapître 0) des commandes gardées dont la valeur dépend d'un message reçu.

---

*) Attention là à une ambiguité possible, un processus peut être complet sans qu'il l'ait lui-même détecté si on s'en tient à la définition stricto-sensu. On supposera quand même que lorsque la logique de l'algorithme l'exige, un processus est capable de diagnostiquer son passage d'un état à un autre. Cette possibilité doit s'exprimer dans la spécification de l'algorithme lui-même.

IV - ALGORITHME DE CONTROLE ET DE TERMINAISON.

IV.1 - Graphes sans circuit:

Dans un graphe sans circuit possédant une "source" (un sommet racine d'une arborescence couvrant les sommets du graphe), la terminaison de l'algorithme distribué est assurée par le fait qu'un tel graphe, comportant une fonction ordinale*), tous les sommets d'un niveau i passent en l'état "terminé" en un temps fini dès lors que tous les sommets de niveau i-1 sont eux-mêmes en l'état terminé. De plus, le graphe possédant une source S, l'ensemble de sommets de niveau 1 de la fonction ordinale du graphe est réduit au singleton {S} qui est complet (puisque $\Gamma(S) = \phi$) et qui passe en un temps fini en l'état terminé. Ceci reste vrai s'il y a plusieurs sources c'est-à-dire, plusieurs sommets de niveau 1. (i.e. plusieurs sommets y tels que $\Gamma^{-}(y)=\phi$).

Sur la Figure 2, les processus associés au sommet de niveau 1 sont complets, ils ne peuvent en effet recevoir de message ; par conséquent une modification des valeurs prises par leurs variables locales ne peut être due qu'à leur algorithme séquentiel propre.

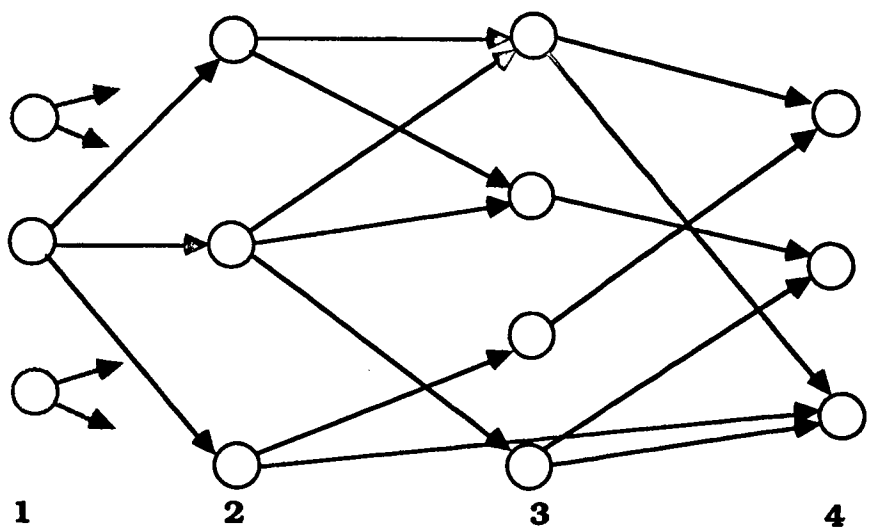


Figure 2

*) On dit qu'un graphe orienté possède une fonction ordinale si, et seulement si, il existe une numérotation possible des sommets telle que, pour tout arc, le numéro du sommet origine de l'arc soit inférieur à celui du sommet extrémité. Ceci implique que dans un tel graphe, on peut définir un préordre sur l'ensemble des sommets, l'ensemble des classes du préordre (chaque classe étant aussi appelée niveau du graphe) étant quant à lui strictement ordonné.

Par conséquent les sommets de niveau 1, étant complets peuvent passer en état fermé et de l'état fermé en l'état terminé. Il s'ensuit que les sommets du niveau i+1 peuvent passer en l'état complet. Et par propagations successives, les processus associés aux sommets de niveau q-1 peuvent passer en l'état terminé, ceux du niveau q en l'état complet, (qui se confond alors avec l'état fermé puisque $\Gamma^+(x) = \phi$ pour ces processus) donc à l'état terminé ; par conséquent l'algorithme distribué se termine en un temps fini.

On peut alors énoncer le théorème :

Théorème :

Une condition nécessaire de terminaison d'un algorithme distribué sur les sommets d'un graphe orienté connexe sans circuit possédant une source, est que la source passe dans l'état "terminé".

On peut généraliser ce résultat pour tout graphe sans circuit, le niveau 1 étant alors composé de plusieurs processus.

Le théorème précédent devient alors :

Corollaire :

Une condition nécessaire de terminaison d'un algorithme distribué sur les sommets d'un graphe orienté connexe et sans circuit est que tout processus X_i associé à un sommet x_i tel que :

$$\Gamma^-(x_i) = \phi$$

passe dans l'état terminé en un temps fini.

Remarque :

Les conditions énoncées ci-dessus ne sont pas suffisantes car la terminaison de chaque processus pris individuellement dépend aussi de l'algorithme séquentiel qu'il comporte. Si on admet que, dès lors qu'il est complet un tel processus passe en l'état "terminé" en un temps fini, ce qui est généralement le cas*) alors les conditions ci-dessus sont aussi suffisantes.

---

*) à l'exception des centraux téléphoniques.

IV.2 - Cas d'un graphe avec circuits:

Supposons qu'un graphe contienne un circuit $x_1, x_2, ..., x_q$. A chaque sommet étant associé un processus $X_1, X_2, ..., X_q$.

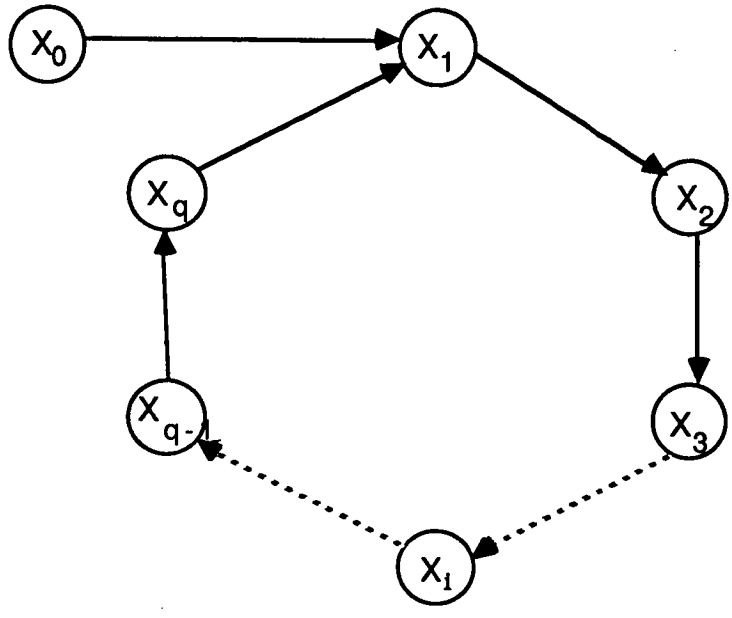


Figure 3

La complétude du processus X_1 est liée à la fermeture des processus X_0 et X_q. La fermeture de X_q est liée à celle de X_{q-1} qui est liée à celle de X_{q-2} qui est liée à celle de ... X_i qui est liée à celle de X_{i-1}... qui est liée à celle de X_2 qui est liée à celle de X_1.

Tant que X_1 n'est pas fermé, il est susceptible d'envoyer des messages vers X_2, donc par voie de conséquence X_q peut recevoir des messages qui sont la conséquence de ceux envoyés par X_1. X_q peut alors envoyer à X_1 des messages qui sont eux mêmes la conséquence des précédents. Il y a là un risque de "bouclage". Pour que chaque processus du circuit puisse passer en l'état complet, puis fermé, il est nécessaire que l'un au moins d'entre eux puisse passer en l'état fermé (prenons X_1) et pour ce faire, il faut qu'il ait détecté les circuits auxquels il appartient ; d'où

Proposition :

Une condition nécessaire pour qu'un processus associé à un sommet d'un circuit d'un graphe puisse passer dans l'état complet est que l'un au moins des processus associés aux sommets du circuit ait identifié ce circuit.

Cette condition n'est bien sûr pas suffisante, un processus pouvant faire partie de plusieurs circuits.

Dans un circuit, la complétude de tout processus est fonction de la fermeture des prédécesseurs dans le même circuit, par conséquent, on peut énoncer :

Proposition :

Une condition nécessaire de terminaison d'un algorithme distribué sur les sommets d'un graphe orienté connexe est que, dans chaque circuit il y ait au moins un processus qui passe de l'état complet à l'état terminé en un temps fini.

V - CONCLUSION

Des résultats précédents, on déduit que, sous certaines conditions liées à la nature du problème (et donc de l'algorithme distribué utilisé pour le résoudre), la connaissance des circuits élémentaires par l'un au moins des sommets*) que contient le réseau permet le contrôle de la terminaison de l'algorithme distribué.

L'équation de point fixe donne le concept théorique qui permet de construire l'algorithme d'*apprentissage* de sa terminaison par un processus. On peut généraliser facilement la problématique à la détection d'états stables dans un réseau distribué. En effet, il suffit de faire abstraction de l'aspect "terminaison" pour pouvoir adapter les algorithmes à des détections d'états stables. Ainsi en est-il dans [C,J,S] du problème de la détection et de l'identification des processus qui sont en interblocage dans un réseau gérant des transactions. En fait, il s'agit de mettre en évidence les processus qui sont sur un même circuit, et qui sont en attente de déblocage de leur successeur.

Par conséquent, dans la deuxième partie, nous allons étudier un algorithme d'identification des-dits circuits par l'un au moins des processus associés aux sommets contenus dans chacun d'eux.

---

*) On fera souvent l'abus de langage qui consiste à identifier un processus au sommet du graphe auquel il est attaché, et vice-versa, lorsque celà n'entraîne aucune ambigüité pour le propos.

2ème PARTIE

La première partie nous a permis de mettre en évidence certaines conditions nécessaires pour la validation des communications interprocessus, Ces conditions font apparaître la nécessité d'identifier les circuits élémentaires d'un graphe, et plus précisément les composantes fortement connexes.

Dans un environnement distribué, nous considérerons les composantes fortement connexes comme étant identifiées dès lors que pour chaque sommet du graphe, celui-ci aura identifié :

- a s'il fait partie d'une composante fortement connexe non réduite à lui-même, et si oui,

- b quels sont ses prédécesseurs immédiats dans cette composante fortement connexe, (les éléments de la composante fortement connexe qui sont aussi éléments de $\Gamma^{-}(x)$).

On notera G = (X,U) le graphe associé au réseau distribué, et par abus de langage, on parlera indifféremment de sommet ou de processus, Par contre, on distinguera les concepts d'arête et d'arc, l'une pouvant être utilisée dans les deux sens, l'autre étant un concept orienté (on peut alors aussi, assimiler un arc à une arête unidirectionnelle). On supposera aussi de plus que pour 2 processus P_i et P_j ;

$$i \neq j \Leftrightarrow P_i \neq P_j.$$

I - LES COMMUNICATIONS

Les communications se font de processus à processus, dans le sens des arcs uniquement, le système d'acheminement des messages, ou système postal, est supposé posséder les propriétés décrites en première partie.

II - ETATS D'UN PROCESSUS

Nous avons décrit dans la première partie les quatre grands états logiques que peut prendre un processus, "En cours", "Complet", "Fermé","Terminé". Pour le problème qui nous préoccupe ici particulièrement, il n'y a pas lieu de distinguer l'état "fermé" de

l'état "terminé"*) et on peut également identifier les états "complets" et "fermé". Par contre, dans l'état "en cours", il nous faut distinguer différents états, ce qui nous conduit à l'ensemble d'états suivants :

1) **Naif** : Le processus peut envoyer ou recevoir des messages, il n'a détecté aucune composante fortement connexe autre que celle réduite à lui-même dont il fasse partie. Il n'a pas détecté non plus qu'il ne faisait partie d'aucune composante fortement connexe non triviale. C'est l'état initial des processus.

2) **Averti** : En ce cas, le processus a été averti par l'un de ses prédécesseurs qu'il fait partie d'un circuit au moins.

3) **Entrée d'un circuit** : Un processus qui détecte lui même (c'est-à-dire sans avoir recu de message d'avertissement) qu'il fait partie d'un circuit, se déclare "entrée d'un circuit".

4) **Complet** : Un processus sera dit "complet" (et s'identifiera comme tel) lorsque :

> Soit il aura identifié qu'il ne fait partie d'aucune composante fortement connexe non réduite à lui-même,
>
> Soit il aura identifié ses prédécesseurs dans la composante fortement connexe dont il fait partie.

Le graphe des états et des transitions est alors le suivant :

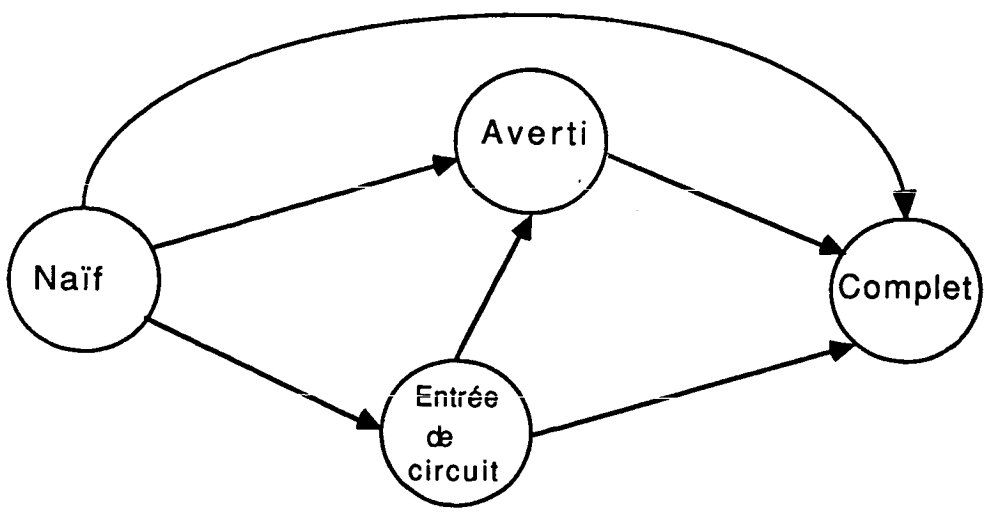


Figure 1

*) En effet, le but assigné à un processus est justement, pour cet algorithme, de détecter qu'il est dans l'état "fermé", il a alors terminé son calcul, ce qui fait que dans ce cas (parce qu'il n'y a pas d'autre calcul à faire) les états "fermé" et "terminé" se confondent.

III - DESCRIPTION INFORMELLE DE L'ALGORITHME

On appelle antécédent, (resp. descendant), d'un sommet x, un sommet y tel qu'il existe un chemin de y à x, (resp. de x à y).

Pour qu'un processus fasse partie d'un circuit, une condition nécessaire et suffisante est qu'il soit à lui-même l'un de ses propres antécédents.

$$x \in A(x)$$

avec

$$A(x) = \Gamma^{-}(x) \cup \Gamma^{-}(\Gamma^{-}(x)) \cup \ldots \cup \Gamma^{-}(\Gamma^{-}(\ldots\Gamma^{-}(x))) \ldots)$$

c'est à dire qu'en fait, A(x) est la fermeture transitive de x.

Dans l'algorithme, un processus "calcule" en fonction des messages qu'il reçoit de ses prédécesseurs et d'après les informations qu'il a ainsi acquises, il peut envoyer un message à ses successeurs (ou à certains d'entre eux).

3.1. Mot circulant :

Un processus "connait" ses prédécesseurs et ses successeurs, mais pas les prédécesseurs de ses prédécesseurs (qu'on a appelé "antécédents"), ni les successeurs de ses successeurs. Pour qu'un processus soit informé du nom de ses antécédents, on utilise un mot circulant. C'est un mot qui est répercuté de processus en processus (c'est une généralisation de la notion de jeton).

Lorsqu'un processus recoit un mot circulant, celui-ci lui est transmis par l'un de ses prédécesseurs et il contient, outre l'identificateur dudit prédécesseur, les identificateurs de tous les antécédents par lesquels il a transité ; le processus ne retransmettra, le cas échéant, le mot circulant qu'après y avoir adjoint son propre identificateur, en queue, de façon à ce que le mot soit une suite ordonnée d'identificateurs, la dernière lettre du mot identifiant le dernier processus ayant réémis le mot circulant.

Ainsi de proche en proche, lorsqu'un processus z recevra un mot circulant de la forme abcd ... γ, il sera informé que le message correspondant a transité , dans l'ordre par les processus a,b,c,d, ..., γ, le processus a étant à l'initiative du message, et γ étant le prédécesseur de z qui lui a envoyé le mot circulant. De cette définition du mot

circulant, on déduit la propriété essentielle qui permet à un processus de s'identifier comme faisant partie d'un circuit donné :

Lemme :

Pour un processus x recevant un mot circulant C, si $x \in C$, alors x est dans un circuit.

De plus, la formation du mot circulant implique, par construction du mot, l'ordre des identificateurs sur celui-ci, et on peut donc énoncer :

Théorème 1 :

Si un processus y reçoit pour mot circulant le mot A y B (A et B étant des sous-mots), alors :

1) y fait partie d'un circuit,

2) ce circuit est composé des processus dont les identificateurs figurent dans le sous-mot B.

Le mot circulant transmis par un processus x à l'un de ses successeurs, soit par exemple y, contient les identificateurs des sommets antécédents de x par lesquels a transité le message pour arriver jusqu'en y. Ce mot circulant est donc de la forme A x, où A est un sous-mot contenant des noms de sommets antécédents à x. Par conséquent, si le mot circulant transmis par x à y est de la forme D y B, celà signifie que y est sur un chemin y B y, par conséquent ce chemin est un circuit (son origine et son extrémité sont confondues).

Corollaire :

Si un processus y fait partie d'un circuit, il recevra, au bout d'un temps fini (car le réseau est supposé fini) un circulant de la forme A y B ; (A et B étant des sous-mots et A étant éventuellement vide).

Remarque :

Dans le cas où l'on admet que le graphe puisse avoir des boucles, (ce qui physiquement n'a pas grand sens), alors le sous-mot B peut aussi être éventuellement vide.

Passage en l'état "entrée de circuit", procédure d'avertissement.

Si y fait partie d'un circuit y, β_i, β_j, ..., β_p, y, lorsque le processus y recoit un circulant de l'un de ses prédédesseurs, soit A ce circulant, il le répercute à tous ses successeurs, donc en particulier à β_i qui le répercute à β_j, etc... β_{p-i} le répercute à β_p. Le circulant est alors de la forme A y β_i, β_j, ..., β_p. Le processus β_p transmet alors son circulant à tous ses successeurs, donc en particulier à y, lequel se détecte comme faisant partie du circuit. Un processus qui se détecte comme faisant partie d'un circuit se déclare "entrée de circuit" et envoie à son successeur *dans le circuit en question* un message l'avertissant de cet état de fait, c'est la procédure d'avertissement.

Passage en l'état "averti"

Lorsque tous les prédécesseurs d'un processus ont soit émis un message d'avertissement, soit un message comme quoi ils sont "complets" le processus concerné passe alors dans l'état "averti", il transmet un message d'avertissement à tous ses successeurs. Ainsi, le message d'avertissement est répercuté de proche en proche jusqu'à ce que tous les processus d'un même circuit soient dans l'état averti, (ou dans l'état entrée de circuit).

Passage en l'état "complet"

3.2. Complétude :

La terminaison d'un processus est un événement local. Pour qu'un processus x puisse prendre l'initiative de s'arrêter, il faut que deux conditions soient remplies ;

a) que les prédécesseurs de x n'envoient plus jamais de message à x ;
b) que x ait fini son traitement local et n'ait plus de message à envoyer.

(En fait, c'est cette deuxième propriété qui permet ici d'identifier l'état complet à l'état fermé du Chapitre I).

De la condition a), on infère que si un processus x n'attend plus de message de ses prédécesseurs, c'est qu'il possède toute l'information qui lui est nécessaire pour ne plus travailler que localement, c'est à dire pour le problème qui nous concerne, que x ait soit identifié qu'il fait partie d'une composante fortement connexe non triviale et qu'il connait ses prédécesseurs dans celle-ci, soit qu'il a identifié que la composante connexe à laquelle il appartient est réduite à lui même, c'est à dire la composante fortement connexe triviale. On peut donc énoncer :

Théorème 2 :

Si un processus Y a identifié que tous les processus qui sont ses prédécesseurs sont complets, alors Y est complet.

Remarque :

On retrouve là, sous une autre forme, la propriété illustrée par l'équation de point fixe de la première partie.

Un processus y soit fait partie d'une cfc (nous écrirons en abrégé cfc pour composante fortement connexe) non triviale, soit fait partie de la cfc réduite à lui même - Dans le deuxième cas, il y a deux possibilités pour y,

i) y est sur des chemins s,...,y dont le premier sommet s n'a aucun prédécesseur (on dit alors que c'est une source), ou il est lui même une source, et donc se déclare "complet". Si y n'est pas une source, il est à l'intersection de plusieurs chemins de type S_i,...,y ; S_j,...,y ; ...,S_p,...,y (les S_i, S_j, ..., S_p étant des sources elles s'identifient comme étant "complet", et en informent de proche en proche leurs successeurs) et chacun des prédécesseurs de y étant "complet", y peut passer dans l'état complet, il ne fait partie d'aucune cfc non triviale, ses prédécesseurs non plus, ils sont donc en mesure d'en transmettre l'information à y qui peut donc se déclarer complet.

Remarque :

On est alors dans le cas simple des graphes sans circuit décrit en première partie, à partir du § IV.1, obtenus sous graphe partiel déduit du réseau initial en ne conservant que les sommets composants les chemins S_i,...y ; S_j,...,y ; etc...

ii) y est sur un (ou des) chemin(s) dont l'origine est un sommet dans un circuit. Si les prédécesseurs de y se sont identifiés comme complets, c'est que tous les sommets de ce (ces) circuit(s) se sont identifiés comme complets, c'est à dire comme faisant partie d'une cfc et connaissant leurs prédécesseurs dans celle-ci ; par conséquent, de proche en proche les prédécesseurs de y se sont déclarés "complets", et y lui même n'a plus à attendre d'autre information de ces sommets, il peut donc se déclarer "complet" lui-même.

Dans le premier cas, y a pour prédécesseurs les processus Y_i, Y_j, ..., Y_p qui sont supposés complets. Ils possèdent par conséquent l'information comme quoi ils font partie chacun d'une composante fortement connexe (i.e. un circuit), y recevant les circulants émis par ses prédécesseurs, va pouvoir conclure s'il fait ou non partie d'une

composante fortement connexe, et, si oui, il connaitra tous ses prédécesseurs dans celle-ci après qu'il ait recu tous les circulants de tous ses prédécesseurs ainsi que leur message de complétude.

Théorème 3 :

- Un processus Y dans l'état "entrée d'un circuit" et dont tous les prédécesseurs sont soit complets, soit avertis, et sont, par Y, identifiés comme tels, est complet.

Si un prédécesseur z de y est complet, c'est que tous ses prédécesseurs le sont, et alors, soit z fait partie d'une composante fortement connexe C_z , soit non. Si C_z existe, alors y en est informé par z, si C_z n'existe pas, y est informé de toute façon que z ne fait pas partie d'une composante fortement connexe. Si tous les prédécesseurs de y qui sont non "complets" sont avertis, cela signifie qu'ils font tous partie de la même composante fortement connexe que y et qu'ils ont été avertis qu'ils font partie d'au moins un circuit. Par conséquent, soient $Y_i, Y_Z, \ldots, Y_p$ les processus avertis prédécesseurs de y, y fait donc partie de p circuits au moins (dont il n'est pas forcément pour chacun d'eux l'entrée) disjoints par au moins un arc. Puisque ces sommets sont dans un état averti, c'est qu'ils font chacun partie d'un circuit dont un sommet au moins s'est déclaré "entrée de circuit" et a envoyé le message d'avertissement. Par conséquent tous ces sommets appartiennent à des circuits ayant le sommet y en commun ; ils font donc partie de la même composante fortement connexe que y, et sont ses prédécesseurs dans celle-ci, et il ne peut y avoir d'autres tels sommets qui ne finissent par passer dans l'état "averti", les autres passant, en un temps fini dans l'état "complet". Ainsi y a bien détecté faire partie d'une composante fortement connexe, et dans celle-ci, il connait ses prédécesseurs, il a donc acquis toute l'information souhaitée, et il peut donc se déclarer "complet".

Lorsque y se déclare complet, l'information correspondante se répercute sur les successeurs, donc le long des circuits, au fur et à mesure que les prédécesseurs des sommets des p circuits seront "complets" ou "avertis" (pour les processus dans l'état entrée de circuit). Par conséquent tous les processus des composantes fortement connexe non triviales auront identifié leur appartenance à une telle composante fortement connexe et connaitrons leurs prédécesseurs dans celle-ci. Des théorèmes 2 et 3 nous pouvons conclure que

Propriété :

Dès lors qu'un circulant atteint un sommet x d'une composante fortement connexe non triviale, il atteindra tous les autres sommets de cette composante fortement connexe.

Ceci est vrai puisque le circulant est répercuté à partir de x sur tous ses successeurs, lesquels le répercutent eux-même sur leurs successeurs, etc, et comme dans une composante fortement connexe il y a un chemin de tout sommet à tout autre, le circulant atteindra bien tous les sommets de la composante fortement connexe. De même on peut énoncer pour les mêmes raisons :

Corollaire :

La procédure d'avertissement est non bloquante*).

En effet, le circulant est répercuté par un sommet à ses successeurs tant que le dit sommet ne se reconnait pas comme ayant déjà été atteint par ce circulant (i.e. tant que son identificateur ne figure pas dans le circulant). Par conséquent, au bout d'un temps fini (la composante fortement connexe comportant un nombre fini de sommets), sur tout circuit de la composante fortement connexe, le circulant atteindra une deuxième fois au moins l'un des sommets. Le sommet ainsi atteint recevra un circulant contenant son identificateur, il se déclarera entrée de circuit, et enverra un message d'avertissement à son successeur sur le circuit considéré. Pour que ce sommet (le successeur), ou l'un de ses successeurs dans le circuit, ne répercute pas le message d'avertissement, il faut que le sommet en question soit en attente, d'un message d'avertissement d'un de ses prédécesseurs, ou d'un message de complétude (ou de l'un et de l'autre, mais sur deux sommets différents).

Considérons donc un tel sommet "bloquant", soit il fait partie d'un circuit, soit non. S'il fait partie d'un circuit, alors en un temps fini, l'un des sommets dudit circuit enverra un message d'avertissement qui atteindra le sommet considéré (le nombre de circuits étant fini, on peut réitérer le raisonnement autant de fois qu'il le faudra), et par conséquent, ce sommet cessera d'être bloquant. Si le sommet "bloquant" ne fait pas partie d'un circuit, il ne fait pas partie de la même cfc que le sommet qu'il bloque, il fait donc partie d'un ensemble de chemins dont l'origine de chacun d'eux est une composante fortement connexe, triviale ou non. Si la composante fortement connexe est triviale, elle est réduite à un sommet qui n'a pas de prédécesseur (sinon le sommet en question ne serait pas origine du chemin considéré), un tel sommet se déclare "complet", et par conséquent, en un temps fini, le sommet "bloquant" (ou tout sommet bloquant le sommet "bloquant" - il y a un nombre fini de tels sommets -) verra ses prédécesseurs concernés se déclarer "complets" et par conséquent pourra lui-même se déclarer "complet", il ne sera donc plus bloquant.

---

*) C'est-à-dire qu'elle n'entraîne pas de situation où un ensemble de processus reste indéfiniment en attente sans pour autant que chacun d'eux n'ait pas terminé sont calcul (i.e. soit passé dans l'état complet).

Si à l'origine du chemin contenant le sommet bloquant se trouve une composante fortement connexe non triviale, puisque elle est à l'origine du chemin, c'est qu'elle n'admet aucun arc incident. Par conséquent, lorsque un de ses sommets "entrée de circuit" va envoyer un avertissement à ses successeurs, celui-ci atteindra en un temps fini tous ses prédécesseurs qui seront alors tous dans l'état "averti", l'entrée de circuit pourra alors passer dans l'état "complet", ainsi que tous les sommets du circuit.

- ils ne peuvent être bloqués, puisque le circuit n'admettant pas d'arc incident, tous ses sommets ont leur prédécesseur dans le circuit lui-même -.

Les différents sommets du chemin non éléments de cette composante fortement connexe vont donc voir leur prédécesseur sur ce chemin passer dans l'état "complet".

Si l'un d'eux est bloqué, on est ramené au cas du sommet bloqué, mais cette fois sur un prédécesseur non élément du chemin précédemment étudié, donc sur un chemin disjoint. On peut alors réitérer le raisonnement précédent, et cela n'aura lieu qu'un nombre fini de fois (le graphe étant fini).

Par conséquent le sommet initialement bloqué passera en un temps fini dans un état "complet" ou "averti". Les raisons qui font que le circulant d'un circuit atteint tous les sommets de celui-ci, et que la procédure d'avertissement est non bloquante, entrainent que la procédure de déclaration de complétude est elle même non bloquante, et par conséquent, on peut énoncer :

Propriété :

La procédure de complétude est non - bloquante.

La procédure de complétude étant non bloquante, et le temps de transmission d'un message étant fini, la procédure de complétude se termine en un temps fini, par conséquent l'identification des composantes fortement connexes se fait en un temps fini.

IV - L'ALGORITHME DE BASE

Afin de préciser les idées, dans un premier temps, supposons que le graphe orienté représentatif du réseau considéré possède une arborescence couvrante de racine donnée X_o .

Le processus associé a X_0, soit P_0 envoie un circulant à tous ses fils. Ne serait-ce que par les arcs constituant l'arborescence couvrante, tous les sommets du graphe recevront un circulant contenant l'identificateur p_0 de P_0. En particulier, tout sommet faisant partie d'un circuit recevra un circulant, donc en vertu du théorème 4, tous les circuits élémentaires seront ainsi parcourus par un circulant et par conséquent toutes les cfc seront identifiées (au sens ici défini).

Nous allons examiner quel doit être le comportement d'un processus en fonction de divers paramètres ; son identificateur, l'état dans lequel il est, la nature des messages recus, l'état dans lequel se trouvent ses prédécesseurs, etc...

4.1. Elements de la spécification

Chaque processus i est doté des variables suivantes :

a) l'ensemble de ses successeurs, il est contenu dans le tableau "fils"

b) l'ensemble de ses prédécesseurs (il le connaît implicitement, voir c et d)

c) un tableau booléen compl tel que :

$$\text{compl}(k) = \begin{cases} 1 & \text{le processus k est complet ou n'est pas prédécesseur de i} \\ 0 & \text{sinon (i.e. k est prédécesseur de i } \underline{\text{et}} \text{ n'est pas complet)} \end{cases}$$

d) un tableau booléen av tel que :

$$\text{av}(k) = \begin{cases} 1 & \text{le processus k est averti ou n'est pas prédécesseur de i} \\ 0 & \text{sinon} \end{cases}$$

e) Compl( ) est un tableau booléen tel que :

$$\text{compl}(k) = \begin{cases} 1 & \text{le processus k, prédécesseur de i est complet} \\ 0 & \text{le processus k, prédécesseur de i n'est pas complet} \end{cases}$$

f) état $\in$ {naïf,av,ent,comp}, av signifiant averti; ent, entrée de circuit; comp, complet, l'abréviation ou le mot complet étant utilisés indifféremment l'un pour l'autre.

g) circ qui contient le circulant.

Lorsqu'un processus reçoit un message du type

P_j?? < x,y,z > , x représente le circulant, y l'état de p_j et z ∈ (avertissement,complet, ∅)

y ∈ (naïf,averti,entrée de circuit,complet)

x ∈ M(X)

où M(X) représente l'ensemble des mots formés sur X.

4.2. Etat naif

Si i est dans l'état naïf, on a :

état=naïf → [i ∈ x → [z = ∅ → état:=ent; k ∈ fils ∩ x, p_k!! <x,état,av>
□
z=av → [Λ (compl(ℓ) ∨ av(ℓ))=1 → k ∈ fils ∩ x, P_k!!<x,av,z>;état:=av
ℓ≠j
□
Λ (compl(ℓ) ∨ av(ℓ))=0 → av(j) := 1 *)
ℓ≠j
]
]
□
i ∉ x → [y=complet → [Λ compl(ℓ)=1 → état:=complet
ℓ≠j
□
Λ compl(ℓ)=0 → circ:=x|i ; ∪ :k∈fils, P_k!!<circ,naïf,∅> ;
ℓ≠j k
compl(j):=1
]
□
y≠complet → circ:=x|i ; ∪ :k∈fils, P_k!!<circ,naïf,ϕ>
k
]
]

Si P_i est dans l'état naif et que l'identificateur i fait partie du circulant transmis par le processus j, si il n'y a pas de message d'avertissement (c'est à dire que z= ∅) alors p_i est entrée de circuit (i.e. dans le circuit pour lequel les identificateurs de sommets le composant sont contenus dans le circulant, i est le premier des sommets concernés à s'identifier comme faisant partie dudit circuit) ; il peut alors envoyer un message d'avertissement à son fils dans le circuit détecté (c'est à dire à k ∈ fils ∩ x ; x étant le circulant envoyé par p_j). Si p_i est dans l'état naif et que l'identificateur i ne fait pas partie du circulant, on ne peut conclure que si y = complet, ce qui signifie que le processus j qui a envoyé le message est lui même complet.

---

*) Le processus i ne peut être dans l'état naïf, recevoir un circulant contenant i et un message z de complétude.

Par conséquent, si tous les autres prédécesseurs de i sont dans l'état complet (c'est le test $\underset{\ell \neq j}{\Lambda}$ compl(ℓ) = 1), alors i aussi peut passer dans l'état complet (voir le théorème 2).

4.3. Cas du processus dans l'état "averti"

Dans le cas où i est dans l'état averti, c'est que tous ses prédécesseurs sont soit avertis ou entrée de circuit soit complets et qu'il y en a au moins un d'averti (sinon i serait déjà complet).

Le seul type de message que puisse attendre i est précisément le message de complétude, d'où :

état=averti → [z = complet → [$\underset{\ell \neq j}{\Lambda}$ compl(ℓ) = 1 → état:=complet
□
$\underset{\ell \neq j}{\Lambda}$ compl(ℓ)=0 → compl(j) = 1
]
□
z = av → k ∈ fils ∩ x, P_k!! <x,av,av>
/* i pouvant appartenir à plusieurs circuits, il doit transmettre les messages d'avertissement afférents/*
□
y = naïf → [i ∈ x → état:=ent ; k ∈ fils ∩ x; P_k!! <x,état,av>, circ:=x
□
i ∉ x → x:=x| i; $\underset{k}{\cup}$:k ∈ fils, P_k!! <x,naïf,∅>
]
]

4.4. Cas du processus dans l'état "entrée de circuit"

Dans le cas où i est dans l'état entrée de circuit c'est que :

a) l'un au moins de ses prédécesseurs n'est pas complet

b) l'un au moins de ses prédécesseurs est averti.

Dès lors, si i reçoit un message d'avertissement, celui-ci ne peut provenir que d'un circuit différent de ceux pour lesquels i s'est déclaré entrée de circuit. Ainsi, supposons que sur la figure 2 ci-dessous, a se soit déclaré "entrée de circuit" suite à la réception du circulant pnabcdef ; "a" a alors envoyé un avertissement à b qui l'a envoyé à c ... à f à "a" lui-même qui n'en tient pas directement compte sinon qu'il enregistre :

av(f) := 1.

Issu de p, il y a un circulant qui va devenir pohijklmag, alors h se déclare "entrée de circuit" sur ce circulant et envoie un message d'avertissement à i qui l'envoie à j, etc... à m qui l'envoie à a qui enregistre encore

av(m) = 1

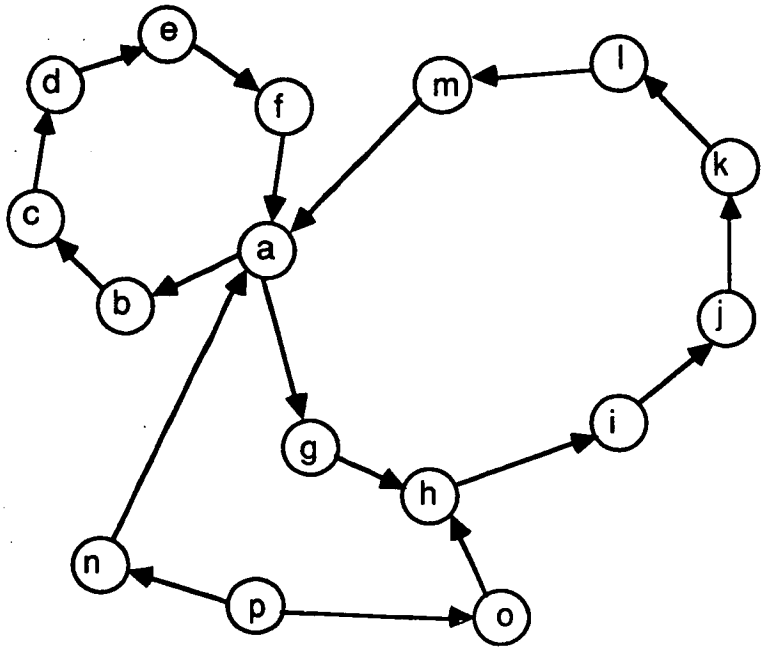


Figure 2

et tous les prédécesseurs de a sont alors complets ou avertis. n est en effet complet puisque son seul prédécesseur p est complet, n'ayant pas lui même de prédécesseurs. O est complet pour la même raison que n.

a passe alors dans l'état complet, provoquant le passage à cet état des processus b et g, b complet entraîne c complet qui entraîne d complet qui entraîne e complet qui entraîne f complet. g et O complets entrainent que h passe à l'état complet puis i, j, k, l, m dans l'ordre et l'algorithme s'arrête.

D'où pour i dans l'état entrée de circuit, la séquence :

état = ent → [y≠naïf → [$\underset{\ell \neq j}{\wedge}$ (compl(ℓ) ∨ av(ℓ)) = 1 → état:=complet
□
$\underset{\ell \neq j}{\wedge}$ (compl(ℓ) ∨ av(ℓ)) = 0 → av(j) := 1
]
□
y = naïf → x:=x|i; $\underset{k}{\cup}$:k ∈ fils , P_k!!<x,naïf,∅>
]

Si le processus j était dans l'état naïf, il faut répercuter le circulant.

En effet, sur la figure ci-dessus, supposons que la détection du circuit abcdef ait été très rapide, et que m soit encore dans l'état naïf et n'ait encore pas reçu de message de l alors que a s'est déjà déclaré "entrée de circuit". m, dans l'état naïf reçoit de l un circulant qu'il répercute sur a qui le répercute sur tous ses fils (voir le cas où z ≠ complet pour état = averti) et seuls ceux qui sont dans l'état "naïf" traitent ce message.

Nous avons fusionné le traitement de y = averti et y = entrée de circuit car, du point de vue de i, c'est pareil, y = entrée de circuit est un cas particulier de y = averti.

4.5. Cas du processus dans l'état "complet", spécification Générale de l'algorithme

Si le processus i est dans l'état "complet", c'est qu'il n'a plus rien à faire, il a détecté s'il faisait ou non partie d'une cfc non triviale, et éventuellement, il connaît ses prédécesseurs dans la cfc (ce sont tous les prédécesseurs lui ayant envoyé un message d'avertissement).

Le processus n'a plus qu'à avertir ses fils de sa complétude et à s'arrêter.

Soit, en récapitulant :

P_i :: état:=naïf; compl( ) := ∅ ; av( ) := ∅ ;
*[pred = ∅ → $\underset{k}{\cup}$: k ∈ fils, P_k!! <i,naïf,∅> ; complet
□
P_j??<x,y,z> → [état = naïf → .... (cf. paragraphe 4.2)
□
état = averti → .... (cf. paragraphe 4.3)
□
état = ent → .... (cf. paragraphe 4.4)
]
□
état = complet → $\underset{k}{\cup}$: k ∈ fils, P_k!! <∅,complet,complet> ; stop
]

Remarque 1 :

En fait l'algorithme spécifié ici est plus général que celui qui consiste à travailler sur un réseau possédant une arborescence couvrante de racine connue. Cet

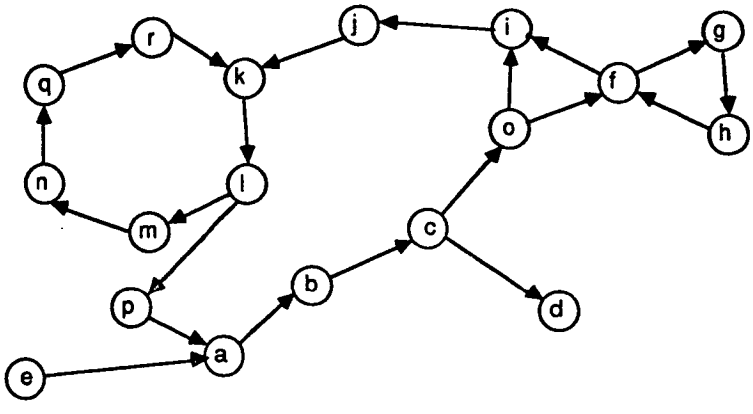
algorithme permet de résoudre le problème dans le cas ou il existe une forêt couvrante, ce qui est toujours réalisé.

Bien entendu le cas de l'arborescence couvrante est ainsi résolu.

Remarque 2 :

Le principe de l'algorithme s'apparente dans la démarche à celle de [MIS], en fait le circulant joue ici le rôle d'un jeton au sens de Misra, et ce circulant porte en lui même l'information sur le circuit parcouru (ce que ne fait pas le jeton de Misra).

4.6. Fonctionnement sur un exemple



Pour construire un exemple représentatif sur des problèmes de recherche de cfc, on prendra un graphe en forme de "cactus", afin de faire apparaître les particularités du problème.

Figure 3

Considérons le graphe de la figure 3 avec, en chaque sommet un processus. Les délais de transmission étant non prévisibles, le calcul sur un tel réseau est non déterministe.

En conséquence de ce non déterminisme, l'exemple de fonctionnement de l'algorithme que nous donnons est un exemple parmi d'autres possibles. On ne peut prédire à l'avance quelles instances de calcul seront effectuées d'une part, et d'autre part on ne peut assurer qu'en appliquant deux fois le même algorithme sur le même réseau, on obtienne le même ensemble d'instances de calcul.

Etape 1

Processus	e	a	b	c
circulant	e	∅	∅	∅
état	naïf	naïf	naïf	naïf
message	a	∅	∅	∅

Etape 2

Processus	e	a	b	
circulant	e	ea		
état	comp	naïf	naïf	naïf
message	a	b		

Etape 3 (on a supprimé de la liste des processus, e qui est complet)

Processus	a	b	c	d
circulant	ea	eab		
état	naïf	naïf	naïf	naïf
message	∅	c		

Etape 4

Processus	a	b	c	d
circulant	ea	eab	eabc	∅
état	naïf	naïf	naïf	naïf
message	∅	∅	e d	∅

Etape 5

Processus	a	b	c	d	o
circulant	ea	eab	eabc	eabcd	eabco
état	naïf	naïf	naïf	naïf	naïf
message	∅	∅	∅	∅	f,i

Etape 6 (on ne fait figurer que les processus pour lesquels "il se passe quelque chose"

Processus	o	f	i
circulant	eabcof	eabcof	eabcoi
état		naïf	naîf
message		i,g	j

Etape 6

Processus	f	i	j	g
circulant	eabcof	eabcofi	eabcoij	eabcofg
état	naïf	naïf	naîf	naîf
message		j	k	h

Etape 7

Processus	j	g	h	k
circulant	eabcofij	eabcofg	eabcofgh	eabcoijk
état	naïf	naïf	naïf	naîf
message	k		f	ℓ

Etape 8

Processus	k	ℓ	f
circulant	eabcofij	eabcoijkl	eabcofghf
état	naïf	naïf	entrée de circuit
message	ℓ	m , p	avertissement à g

Etape 9

Processus	g	ℓ	m	p
circulant	eabcofg	eabcofijkl	eabcoijklm	eabcoijklp
état	avertit	naïf	naïf	naïf
message	avertissement à h	m , p	n	a

Etape 10

Processus	a	h	m	n	p
circulant	eabcoijklpa	inchangé	eabcofijklm	eabcoijklmn	eabcofijklp
état	entrée de circuit	avertit	naïf	naïf	naïf
message	mesage d'av b	av à f	n	q	a

Etape 11

Processus	b	a	f	n	q
circulant	inchangé	eabcofijklpa	inchangé	eabcofijklmn	eabcoijklmnq
état	avertit	entrée de circuit	entrée de circuit	naïf	naïf
message	mesage av à c	<eabcofijklpa, naïf,∅> envoyé à b	0	q	r

en attente de complétude

Etape 12

Processus	b	c	q	r
circulant	eabcofijklpab	inchangé	eabcofijklmnq	eabcoijklmnqr
état	entrée de circuit	avertit	naïf	naïf
message	mesage av à c	av. à o	r	k

Etape 13

Processus	c	k	r	o
circulant	inchangé	eabcoijklmnqrk	eabcofijklmnqr	inchangé
état	avertit	entrée de circuit	naïf	avertit
message	av. à o	av. à ℓ sur le circulant c.à.d fils x=ℓ	k	avertiss. à i car i=fils x

Etape 14

Processus	i	k	ℓ	o
circulant	inchangé	eabcofijklmnqrk	inchangé	inchangé
état	avertit	entrée de circuit	avertit	avertit
message	$\emptyset$	<eabcofijklmnqr k,naïf,$\emptyset$> à ℓ	message à m=fils x	message av à f=fils x

Etape 15

Processus	f	ℓ	m
circulant	inchangé	eabcofijklmnqrkl	inchangé
état	COMPLET	entrée de circuit	avertit
message	z=comp à i et à g	message à m avertissement	mesage av. à n

Etape 16

Processus	i	m	n	g
circulant	inchangé	inchangé	inchangé	inchangé
état	avertit	avertit	avertit	COMPLET
message	av. à j	av. à n	av. à q	comp. à h

Etape 17

Processus	j	n	q	h
circulant	inchangé	inchangé	inchangé	inchangé
état	avertit	avertit	avertit	COMPLET
message	av. à k	av. à q	av. à r	comp. à f

Etape 18

Processus	k	q	r
circulant	inchangé	inchangé	inchangé
état	ent. de circ.	avertit	avertit
message	∅	av. à r	av. à k

en attente d'av.
ou comp. sur r

Etape 19

Processus	k	r
circulant	inchangé	inchangé
état	COMPLET	avertit
message	comp. à ℓ	av. à k

Etape 20

Processus	ℓ
circulant	inchangé
état	COMPLET
message	comp. à m et p

Etape 21

Processus	m	p
circulant	idem	idem
état	COMPLET	COMPLET
message	comp. à n	comp. à a

Etape 22

Processus	a	m
circulant	"	"
état	COMPLET	COMPLET
message	comp. à b	comp. à n

Etape 23

n, qui est complet envoie un message comp à q
et b qui est aussi complet en envoie un aussi à c

Etape 24

q qui est complet envoie "complet" à r ;
c qui est complet envoie "complet" à d et O

Etape 25

r qui est complet envoie "comp" à k ;
d qui est complet ... ;
O qui est complet envoie un message "comp" à f et i

Etape 26

f étant complet et reconnu comme tel par i,
i est complet et envoie un message "comp" à j

Etape 27

j qui est complet envoie un message "comp" à k

TOUS LES PROCESSUS SONT COMPLETS,

l'algorithme est terminé.

4.7. Discussion

On peut imaginer des variantes à cet algorithme, afin de limiter la quantité de messages échangés.

Ainsi lorsqu'un processus est "averti" on pourrait lui faire envoyer un message d'avertissement à tous ses fils. Avec cette façon de faire, un processus averti recevant un message d'avertissement n'aurait pas à retransmettre celui-ci.

Ainsi dans l'exemple précédent, on éviterait des phénomènes comme celui des étapes 8 à 14, où sur le même circuit circulent deux circulants concernant ce circuit et où ce circuit aura deux sommets "entrée de circuit". D'autres modifications encore sont possibles, mais nous avons choisi de donner une version de l'algorithme qui se spécifie le plus simplement possible, sans prendre en compte les problèmes d'implémentation *).

V - CAS GENERAL, GRAPHE QUELCONQUE

Bien que généralement dans un réseau distribué il existe une arborescence couvrante minimale au moins, nous allons envisager le cas où nous ignorons tout de la topologie du graphe de ce point de vue.

En ce cas, l'algorithme précédemment proposé ne saurait convenir. En effet, dans cet algorithme, seuls les sommets "source", c'est à dire ceux qui n'ont aucun prédécesseur initialisent l'exécution de l'algorithme. Dans un graphe quelconque, il n'y a pas forcément de sommet source. Dans un tel contexte, il faut que chaque processus initialise l'algorithme par l'émission d'un circulant et d'un seul.

Pour qu'il n'y ait émission que d'un seul circulant, on crée une commande alternative dont l'une des gardes qu'elle contient est formée par une variable binaire "bin", telle que

$$\text{bin} = \begin{cases} 0 & \text{tant que le circulant n'a pas été émis} \\ 1 & \text{sinon} \end{cases}$$

de telle sorte que lorsque bin = 1, on ne remet jamais cette variable à 0.

La spécification de l'algorithme général est alors :

```
P_i ::      bin:=0 ; état=naïf ; compl( )=∅ , av( ):=∅
*[bin=0 →   ∪ : k ∈ fils, P_k!! <i,naïf,∅> ; bin:=1
□           k
bin=1 →     P_j??<x,y,z> → [état = naïf → ...        (cf. paragraphe 4.2)
                           □
                           état = averti → ...       (cf. paragraphe 4.3)
                           □
                           état = ent → ...          (cf. paragraphe 4.4)
                           ]
□
            état = complet → ∪ : k ∈ fils, P_k!! <∅,complet,complet> ; stop
]                            k
```

*) Une telle implémentation est en cours en OCCAM sur un réseau de transputers.

Avec cette façon de faire, chaque processus émet un circulant, alors qu'en fait il suffirait d'un seul circulant par composante fortement connexe (théorème 4), on a donc redondance de circulants.

Remarque : On peut limiter cette redondance en imposant qu'un processus qui reçoit un circulant ne puisse lui-même être à l'initiative de l'émission d'un circulant s'il ne l'a pas déjà fait.

VI - LE NOMBRE DE MESSAGES ECHANGES

Lorsque tous les processus sont naïfs, ils envoient le circulant sur tous les arcs dont ils sont origine. Tout arc ayant une origine, tout arc est parcouru une fois au moins par un circulant. Dans le cas de l'algorithme sur un graphe possédant une source, chaque circuit est parcouru trois fois,

- une fois par un circulant émis par des processus dans l'état naïf,
- une fois par un message d'avertissement,
- une fois par un message de complétude.

Le nombre de messages émis par un tel algorithme est de l'ordre de :

$$O(m)$$

De même, en ce qui concerne l'algorithme dans le cas général, il consiste schématiquement en la superposition d'autant de fois l'algorithme de base qu'il y a de sommets dans le graphe.

Par conséquent, le nombre de messages émis par l'algorithme général est, dans le pire des cas, de l'ordre de :

$$O(m.n)$$

La remarque du §. V est de nature à limiter très fortement le nmbre de circulants émis.

VII - PARTICULARISATION A LA TERMINAISON DISTRIBUEE

Attention toutefois ; il s'agit ici du nombre de messages nécessaires à la terminaison de *tous* les processus d'un algorithme. Si on en revient à la problématique initiale, à savoir qu'on s'intéresse *pour un processus* à apprendre qu'il a terminé, le nombre de messages est alors en

$$O(p)$$

où p est le nombre d'arcs de la composante fortement connexe contenant le processus considéré. C'est l'aspect le plus intéressant dans le contexte exposé dans la première partie.

Si le but que l'on s'assigne est la détection de la terminaison distribué, sans chercher à acquérir de connaissance sur la topologie du graphe, le circulant devient inutile.

Dans ce cas, suivant les propriétés, connues a priori, du réseau de communication, on peut déduire différents algorithmes.

7.1. Réseau fortement connexe

Si on sait que le graphe représentatif du réseau est fortement connexe, il existe un circuit contenant tous les sommets du graphe, et en initialisant une procédure de jeton circulant (aux lieux et place du mot circulant), on peut détecter l'état de chacun des processus, et en déduire une éventuelle terminaison distribuée selon que tous les processus visités seront restés ou non dans un état passif.

7.2. Réseau quelconque

Si on ne sait rien de la topologie du réseau, on peut lancer un jeton depuis chaque sommet du graphe, en le munissant d'un identificateur (celui du processus associé au sommet qui initialise le jeton).

En s'appuyant sur ces principes, on retrouve deux algorithmes connus de détection de la terminaison (voir [DI,FE,GA] et [MIS]).

Toutefois, l'algorithme de [DI,FE,GA] présuppose la connaissance d'un circuit sur le graphe, ce qui suppose un algorithme pour ce faire et laisse en suspens le problème de la terminaison dudit algorithme (on peut appliquer celui que nous donnons au paragraphe II).

Les inconvénients de cet algorithme ont été partiellement levés par Topor [TOP] mais ce dernier réintroduit une structure de contrôle qui est un arbre couvrant, ce qui présuppose ici encore un algorithme pour ce faire. Nous donnons un tel algorithme dans [LA,RO] et [LA2].

L'algorithme de Misra [MIS] lui ne fait aucune supposition préalable sur la topologie du graphe et en cela est proche des algorithmes donnés aux paragraphes IV et V.

Par ailleurs, on peut faire d'autres hypothèses sur les possibilités et les contraintes liées aux communications. Ainsi, les algorithmes que nous avons donné aux paragraphes IV et V, de même que ceux de [MIS] et [DI,FE,GA], respectent pleinement la contrainte du sens de circulation de l'information liée à l'orientation des arcs. Certains auteurs relachent plus ou moins cette contrainte, et admettent par exemple que certains signaux peuvent circuler sur un arc à contrario de son orientation, des acquits par exemple.

On peut alors modifier simplement l'algorithme du §.IV, en gérant deux compteurs en chaque processus, gérant l'un les émissions de messages, réception d'acquits, l'autre les réceptions de messages, émissions d'acquits.

Les règles d'incrémentation des deux compteurs sont alors les suivantes pour chaque processus :

- envoi d'un message compt1 := compt1 + 1
- envoi d'un acquit compt2 := compt2 - 1
- réception d'un message compt2 := compt2 + 1
- réception d'un acquit compt1 := compt1 - 1

Le compteur compt1 est associé aux envois de messages et aux acquits consécutifs à ces envois de messages.

$$\{\text{compt1} = 0\} \Leftrightarrow \{\text{il y a eu autant d'acquits reçus que de messages envoyés}\}$$

Le compteur compt2 gère lui la réception des messages

$$\{\text{compt2} = 0\} \Leftrightarrow \{\text{On a envoyé autant d'acquits qu'on a reçu de messages}\}$$

En gérant ces compteurs pour détecter la terminaison d'un algorithme sur la racine d'une arborescence couvrante, on retrouve l'algorithme de Dijkstra et Scholten (voir [DI,SC]).

BIBLIOGRAPHIE

[AP,FR] APT K.R., FRANCEZ N., "Modeling the Distributed Termination convention of CSP", ACM Toplas, Vol. 6,3, July 1984, pp. 370-379.

[BO1] BOUGE L., "Symmetric Election in CSP", Rapport CNRS 84-31, CNRS LA 248, LITP, 2 Place Jussieu, 75221 Paris Cédex 05, Juin 1986.

[BO2] BOUGE L., "Repeated snapshots in distributed systems with synchronous communications and their implementation in CSP", Theoretical Comp. Sc. 49, 145-169, 1987.

[CA,RO] CARVALHO O.S.F., ROUCAIROL G., "Assertion, Decomposition and Partial Correctness of Distributed Control Algorithme", in Distributed Computing Systems, Parker Y. and Verjus J.P., ed., Academic Press, 1983, pp. 67-92.

[CAR] CARVALHO O.S.F., "Une contribution à la programmation des systèmes distribués", Thèse d'Etat, 1985, Université Paris VI.

[CH,MI1] CHANDY K.M., MISRA J., "Termination detection for diffusing computations in communicating sequential processes", ACM Trans. pro. Lang. Syst., 4, 1 janvier 1982, pp. 37-43.

[CH,MI2] CHANDY K.M., MISRA J., "Distributed Computation on Graphs : Shortest Path Algorithms", CACM Vol 25 n° 11, pp. 833-837, Nov. 1982.

[CH,LA] CHANDY K.M., LAMPORT L., "Distributed snapshots : determining global states of distributed systems", ACM TOCS, pp. 63-75, 1985.

[C,J,S] CIDON I., JAFFE J., SIDI M., "Local Distributed Deadlock Detection by Cycle Detection and Clustering", IEEE Trans. on Software Engineering, vol. SE13, n°1, January 1987, pp. 3-14.

[DI,FE,GA] DIJKSTRA E.W., FEIGEN W.H.J. and VAN GASTEREN A.J.M., "Derivation of a termination detection algorithm for distributed computations", Inf. Proc. Lett. 16 (5), 1983, pp. 217-219.

[DIJ] DIJKSTRA E.W., "Cooperating Sequential Processes" in Genuys F. (ed.), Programming Languages, Acad. Press, New York, pp. 43-112, 1968.

[DI,SC] DIJKSTRA E.W., SCHOLTEN, "Termination detection for diffusing computations", IPL, n°11, pp. 217-219, Août 1980.

[FRA] FRANCEZ N., "Distributed termination", ACM Trans. pro. lang. syst. 2, 1, Janvier 1980, pp. 42-55.

[FR,RO] FRANCEZ N., RODEH M., "Achieving distributed termination without freezing", IEEE Trans. Soft. Eng. - 8 - (3), 1982, pp. 287-292.

[GAR] GARRETT BIRKOFF, "Lattice Theory", AMS Colloquium Publications, Vol. XXV, Third Edition 1967.

[GE,PU] GELENBE E. et PUJOLLE G., "Introduction aux Réseaux de Files d'attente", éditions Eyrolles et CNET-ENST 1982.

[H,P,R] HELARY J.M., PLOUZEAU N., RAYNAL M., "Calculs d'états globaux remarquables dans un système réparti", Rapport de Recherche n°396, IRISA, Mars 1988.

[HOA] HOARE C.A.R., "Communicating Sequential Processes", CACM, August 1978, Vol. 21, n° 8, pp. 666-677.

[LA,RO] LAVALLEE I., ROUCAIROL G., "A Fully distributed (minimum) spanning tree algorithm", Inf. Proc. Lett., Août 1986.

[LA1] LAVALLEE I., "Reconnaissance du caractère distributif d'un treillis fini", CRAS, Paris, t.282, Série A, pp. 1339-1341, 21 juin 1976.

[LA2] LAVALLEE I., "Contribution à l'algorithmique, parallèle et distribuée", Thèse d'état, Université Paris XI (Orsay), Juin 1986.

[MAT] MATTERN F., "Algorithms for distributed termination detection", Distributed Computing, Vol. 2, n° 3, pp. 161-175, 1987.

[MIS] MISRA J., "Detecting Termination of Distributed Computation using Markers", Proc. of the 2d annual ACM Symposium on Principles of DC. Montréal, August 1983, pp. 290-294.

[RAN] RANA S.P., "A Distributed Solution of the Distributed Termination Problem", Inf. Proc. Letters, Vol. 17, July 83, pp. 43-46.

[RAY] RAYNAL M., "Algorithmes distribuées & protocoles", Eyrolles 1985.

[TOP] TOPOR R.W., "Termination detection for distributed computations", Inf. Proc. Lett. 18, 1984, pp. 33-36.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique